YOUR PERL IS SLOW

(and it does not have to be)

TIMTOWTDI
TIOOFWTDI

- There is more than one way to do it, but there is only one fastest way to do it.
- If you find the fastest way, nobody will be able to read your code and everyone reading your code will probably hate you.
- We need to find something in between (unless of course nobody is reading your code).

WHEN AND WHAT DO YOU OPTIMIZE?

- It is almost always the most advantageous to start with the slowest part of your program because you will spend the least amount of time on it for the most amount of gain. There are many good perl tools to help us find where that is and make it faster:
  - Devel::NYTProf, Benchmark, Data::Dumper
- The more time you spend optimizing, the more you will see diminishing returns.

OPTIMIZING IS A TRADE-OFF

- Make sure you make a backup of your original code before you start optimizing so you can use it to compare the two.
- You will invariably break something when you optimize unless you have 100% test coverage (which probably is never possible in the real world).
- Consider that you may be trading maintainability for speed.

OPTIMIZATION. THE ARGUMENT

- Should I optimize XYZ?
  - Ask yourself:
    - How often is this code actually executed?
    - Does it matter if subroutine X is slow (is this something that runs in the background or an AJAX response?)
    - How much pain am I going to cause the people I work with by changing this?
    - Could I be spending my time better elsewhere?
    - Can I just throw more hardware at the problem?

TELL ME WHY MY PROGRAM IS SLOW.

- Devel::NYTProf - Powerful fast feature-rich perl source code profiler
- Benchmark - benchmark running times of Perl code
- Data::Dumper - on \%INC , and \%main::

DEVEL::NYTPROF CANNOT TELL YOU WHO YOU ARE

- Devel::NYTProf will show you where your perl program is slow, but it does not tell you how to fix it.
- Once you know where it is slow, you will know where to direct your efforts to make it faster.
- This may involve a bit of trial an error, google searching, or general hard thinking.

DEVEL::NYTPROF BASICS

- perl -d:NYTProf script.pl
- nytprofhtml (nytprofcsv exists, but not covered here)
- Open up nytprof/index.html in your web browser
  - If you are running it on a server you can do
    - rsync -avz USER@SERVER:/PATH/ nytprof-project/
    - ** CAREFUL ** the nytprof dir will have a complete copy of your source code.

CONTROLING NYTPROF DATA COLLECTION

- export NYTPROF=trace=2:start=init:file=/tmp/nytprof.out
- trace=2
  - the trace level (more detailed output)
- start=init
  - skip profiling BEGIN (and use)
- file=/tmp/nytprof.out
  - Where to store the output (.$$ is append automatically if we fork)

EXAMPLE OUTPUT (FOLLOW THE RED)

Profile of ./quotacheck for 137s (of 141s), executing 657080 statements and 190536 subroutine calls in 77 source files and 18 string evals.

Jump to file...

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
5120	1	1	48.3s	48.3s	Cpanel::Email::DiskUsage::CORE:stat (opcode)
9416	2	1	41.9s	41.9s	main::CORE:ftis (opcode)
5097	1	1	35.6s	35.6s	Cpanel::Email::DiskUsage::CORE:read (opcode)
5116	1	1	2.08s	2.08s	Cpanel::AdminBin::CORE:ftis (opcode)
32	4	1	1.36s	1.36s	Cpanel::ContactInfo::CORE:readline (opcode)
5120	1	1	1.15s	89.4s	Cpanel::Email::DiskUsage::_getdiskused
4171	2	1	791ms	1.19s	Cpanel::Config::LoadCpUserFile::_load
5120	1	1	495ms	659ms	Cpanel::Email::Maildir::_find_maildirsize_file
5116	1	1	470ms	2.96s	Cpanel::AdminBin::check_cache_item
4175	5	5	234ms	234ms	Storable::pretrieve (xsub)
5116	1	1	209ms	345ms	Cpanel::AdminBin::get_cache_dir
5120	1	1	189ms	89.5s	Cpanel::Email::DiskUsage::get_disk_used
24	1	1	165ms	165ms	main::CORE:ftbinary (opcode)
1	1	1	148ms	181ms	Cpanel::ContactInfo::fetch_contactinfo
4163	1	1	148ms	1.33s	Cpanel::Config::LoadCpUserFile::load

See all 836 subroutines

EXAMPLE OUTPUT (FOLLOW THE RED)

```
# spent 48.3s within Cpanel::Email::DiskUsage::CORE:stat which was called 5120 times, avg 9.44ms
# 5120 times (48.3s+0s) by Cpanel::Email::DiskUsage::_getdiskused at line 213, avg 9.44ms/call
sub Cpanel::Email::DiskUsage::CORE:stat; # opcode
```

Line	State ments	Time on line	Calls	Time in subs	Code
213	5120	48.4s	5120	48.3s	`my ( $maildir_size_file_size, $maildir_size_file_mtime ) = (` `# spent 48.3s making 5120 calls to Cpanel::Email::DiskUsage`

- We can tell that we have spent 48.3s making 5120 calls to Cpanel::Email::DiskUsage::CORE::stat.

This is a call to perl's internal stat function, which is really just a system call to stat

LETS DO IT AGAIN JUST TO BE SURE

Profile of ./quotacheck for 6.93s (of 10.9s), executing 646199 statements and 185395 subroutine calls in 77 source files

Jump to file...

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
5120	1	1	1.07s	3.05s	Cpanel::Email::DiskUsage::_getdiskused
4155	1	1	801ms	1.19s	Cpanel::Config::LoadCpUserFile::_load
5120	1	1	476ms	628ms	Cpanel::Email::Maildir::_find_maildirsize_file
5116	1	1	442ms	792ms	Cpanel::AdminBin::check_cache_item
4160	6	6	241ms	241ms	Storable::pretrieve (xsub)
5116	1	1	202ms	258ms	Cpanel::AdminBin::get_cache_dir
5120	1	1	181ms	3.23s	Cpanel::Email::DiskUsage::get_disk_used
4155	1	1	151ms	1.34s	Cpanel::Config::LoadCpUserFile::load
1	1	1	147ms	154ms	Cpanel::ContactInfo::fetch_contactinfo
27446	4	1	134ms	134ms	Cpanel::Email::Maildir::CORE:match (opcode)
5097	1	1	124ms	124ms	Cpanel::Email::DiskUsage::CORE:read (opcode)
9311	16	13	94.6ms	94.6ms	IO::Handle::DESTROY
4155	1	1	89.2ms	1.43s	Cpanel::Config::LoadCpUserFile::loadcpuserfile
15362	4	1	87.9ms	87.9ms	Cpanel::Email::DiskUsage::CORE:match (opcode)
9416	2	1	77.5ms	77.5ms	main::CORE:ftis (opcode)

See all 814 subroutines

WHAT HAPPENED?

- The first run took over 100 seconds; the second run took under 10 seconds.
  - Almost all the overhead in this program is disk i/o. On the second run Linux cached our reads/opens so it was much faster.
- We can now see where our perl code is slow (***_getdiskused***) instead of what is taking time to read from the disk.
- In this instance we can get the most gains by redesigning how data is stored on the disk instead of optimizing the perl code. The slowest part is really less the 1% of the time. We should probably focus our efforts there if we think its worth the time.
- This code has already been though an optimization cycle.

ALL PROFILERS LIE

- All profilers must record timing information. Running code though a profiler will make it slower.
- There may be race conditions in your code that make it behave differently when profiled.
- The overhead of the profiler makes something appear slower then it really is.
- If you are doing lots of disk reads, you must consider that the os may cache your file opens/reads/etc and the profile will seem much faster the second time around. (sometime you need to reboot to ensure a pristine run)

SOMETHING WE CAN SPEED UP IN A SNAP

Profile of what_is_a_hash.pl for 16.6s (of 16.8s), executing 26140 statements and 7948 subroutine calls in 62 source files and

Jump to file...

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
1	1	1	16.2s	16.2s	main::print_if_we_have_it
466	1	1	91.8ms	123ms	DateTime::Locale::_register
4004	1	1	54.5ms	54.5ms	main::CORE:print (opcode)
468	2	2	28.5ms	28.5ms	Params::Validate::_validate (xsub)
1	1	1	25.5ms	57.3ms	DateTime::Locale::add_aliases
1	1	1	18.8ms	280ms	main::BEGIN@6
423	2	1	16.5ms	16.5ms	Params::Validate::_validate_pos (xsub)
422	1	1	15.3ms	31.8ms	DateTime::Locale::_registered_id
1	1	1	13.8ms	14.3ms	DateTime::Locale::BEGIN@11
1	1	1	11.3ms	76.1ms	main::BEGIN@3
12	12	10	10.8ms	17.3ms	base::import (recurses: max depth 2, inclusive time 5.20ms)
28	28	14	10.3ms	15.2ms	Exporter::import
1	1	1	9.43ms	132ms	DateTime::Locale::register
1	1	1	5.64ms	20.7ms	IO::Socket::SSL::BEGIN@17
1	1	1	5.34ms	9.12ms	DateTime::Locale::BEGIN@10

See all 1284 subroutines

THERE IS THE SLOWDOWN

We are spending all of our time in sub print_if_we_have_it

Line	State ments	Time on line	Calls	Time in subs	Code

14					`# spent 16.2s (16.2+54.5ms) within main::print_if_we_have_it whi`
					`#  once (16.2s+54.5ms) by main::RUNTIME at line 12`
					`sub print_if_we_have_it {`

15	1	2.21ms			`  my @product_array = ( 'fried cheese', ( 0 .. 10000 ), 'sugar`
16	1	1.92ms			`  for ( 0 .. 1000 ) {`
17	1001	16.3ms			`  foreach my $want ( @{ $_[0] } ) {`
18	4004	16.2s	4004	54.5ms	`  print "We " . ( ( grep { $_ eq $want } @product_arra` `  # spent  54.5ms making 4004 calls to main::CORE:prin`
19					`  }`
20					`  }`
21					`}`

Searching a large array can be slow.

ITS SLOWWWW HOW DO I FIX IT?

- There are generally two approaches, the first one is the more drastic approach:
    - 1) Re-think how your code works. Take a completely new approach to the problem:
        - Rewrite part of your code in C / XS / Inline
        - Are you suffering from startup/shutdown time problems?
            - Consider mod_perl, speedycgi, wrapping up your code into a server, or using perlcc

ITS SLOWWWW HOW DO I FIX IT?

- 2) The less drastic approach : rewrite the little bits:
  - Do you really need those loops?
  - Are you doing lots of operations and then throwing it away?
  - Can you trade memory for speed and do you really want to?
  - Read NCLARK's presentation titled "When perl is not quite fast enough." It has been one of the most useful bits perl documentation I have ever read: http://www.flirble.org/~nick/P/Fast_Enough/
  - map, pack, and grep your code to speed

SPEEDING IT UP

- We can trade memory for speed by using a hash, or some type of external database.
- For example, here is the new function:

```
sub print_if_we_have_it {
    my %product_hash = map { $_ => undef } @{ $_[0] };    #trade memory for speed
    for ( 0 .. 1000 ) {
        foreach my $want (@customer_wants) {
            print "We " . ( exists $product_hash{$want} ? 'do' : 'do not' ) . " have '$w
        }
    }
}
```

MUCH BETTER

Profile of ahh_thats_a_hash.pl for 561ms (of 774ms), executing 26141 statements and 7948 subroutine calls in 62 source

Jump to file...

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
1	1	1	123ms	191ms	main::print_if_we_have_it
466	1	1	88.6ms	118ms	DateTime::Locale::_register
4004	1	1	67.7ms	67.7ms	main::CORE:print (opcode)
468	2	2	27.5ms	27.5ms	Params::Validate::_validate (xsub)
1	1	1	24.7ms	55.5ms	DateTime::Locale::add_aliases
1	1	1	18.6ms	272ms	main::BEGIN@6
423	2	1	16.1ms	16.1ms	Params::Validate::_validate_pos (xsub)
422	1	1	14.8ms	30.9ms	DateTime::Locale::_registered_id
1	1	1	13.6ms	14.0ms	DateTime::Locale::BEGIN@11
1	1	1	11.2ms	75.3ms	main::BEGIN@3
12	12	10	10.6ms	17.0ms	base::import (recurses: max depth 2, inclusive time 5.06ms)
28	28	14	9.83ms	14.6ms	Exporter::import
1	1	1	9.15ms	128ms	DateTime::Locale::register
1	1	1	5.54ms	20.5ms	IO::Socket::SSL::BEGIN@17
1	1	1	5.31ms	9.03ms	DateTime::Locale::BEGIN@10

See all 1284 subroutines

HASHES: ONE OF THE BEST FEATURES OF THE LANGUAGE

Line	State ments	Time on line	Calls	Time in subs	Code

14					`# spent 191ms (123+67.7) within main::print_if_we_have_it which was`
					`#  once (123ms+67.7ms) by main::RUNTIME at line 12`
					`sub print_if_we_have_it {`

15	1	1.90ms			`  my @product_array = ( 'fried cheese', ( 0 .. 10000 ), 'sugar co`
16	1	27.4ms			`  my %product_hash = map { $_ => undef } @product_array;  #sacr`
17	1	9.22ms			`  for ( 0 .. 1000 ) {`
18	1001	14.7ms			`  foreach my $want ( @{ $_[0] } ) {`
19	4004	138ms	4004	67.7ms	`  print "We " . ( exists $product_hash{$want} ? 'do' : 'd` `  # spent  67.7ms making 4004 calls to main::CORE:print,`
20					`  }`
21					`  }`
22					`}`

We went from 16.2s to 138ms on this line by switching to a hash at the expense of some memory.

NYTPROF GOTCHAS

- Make sure you are not closing random file handles in your perl code. NYTProf writes to nytprof.out*, and you if close off those fds you won't get any profile data.

- Make sure your code actually does run faster. Time how long a run of your code takes before and after your changes.
  - Use Benchmark
  - Tools such as time, ps, watch, etc (these are not very exact)

ROUGH VERIFICATION WITH TIME

BEFORE

```
$ time perl what_is_a_hash.pl > /dev/null

real 0m16.810s

user 0m16.793s

sys 0m0.012s
```

AFTER

```
$ time perl ahh_thats_a_hash.pl >/dev/null

real 0m0.214s

user 0m0.208s

sys 0m0.008s
```

VERIFY WITH BENCHMARK

- Extract only the code you need to test.
- Use Benchmark::cmpthese:
  - *cmpthese($count, {*
    'before' => sub { ...before code... },
    'after' => sub { ...after code... },
    });

BENCHMARK TEST CODE

```perl
use Benchmark ();
my @customer_wants = ( 'fried cheese', 'health food', 'sugar cookies', 'a new car' );
my @product_array = ( 'fried cheese', ( 0 .. 10000 ), 'sugar cookies', ( 10002 .. 20000 ), 'health food' );
my %product_hash = map { $_ => undef } @product_array;    #sacrafic memory for speed
Benchmark::cmpthese(
    5000,
    {
        'before' => sub { print_if_we_have_it_before( \@customer_wants ); },
        'after'  => sub { print_if_we_have_it_after( \@customer_wants ); },
    }
);
sub print_if_we_have_it_after {
    foreach my $want ( @{ $_[0] } ) {
        print STDERR "We " . ( exists $product_hash{$want} ? 'do' : 'do not' ) . " have '$want'!\n";
    }
}
sub print_if_we_have_it_before {
    foreach my $want ( @{ $_[0] } ) {
        print STDERR "We " . ( ( grep { $_ eq $want } @product_array ) ? 'do' : 'do not' ) . " have '$want'!\n";
    }
}
```

IT REALLY IS FASTER

- ```
  $ perl benchmark.pl  2>/dev/null
                (warning: too few iterations for a reliable count)
              Rate  before   after
  before    59.0/s      --   -100%
  after   125000/s 211800%
  ```

- We really do not need to do more than 5000 iterations to know the new code is much faster.

- If your numbers are much closer, you should make sure you get a reliable count.
  ```

USING DATA::DUMPER TO FIND IMPORT BLOAT

- Consider the sample script on the right:
- It is all ***use*** Statements
- It took 0.205s seconds to run
- *use X* is really just
  - *BEGIN { require X; import X; }*
- It is not uncommon to see a small script spend more time loading modules then executing the main code.

```
use IO::Socket::SSL;

use IO::Handle;

use Net::SSLeay;

use DateTime;

use IO::Compress::Gzip;

use DateTime::Locale::Catalog;

use POSIX;

use Data::Dumper;
```

IMPORTS FROM USE KILL YOUR STARTUP TIME

```
print Data::Dumper::Dumper(\%main::);

$VAR1 = {
          'setuid' => *::setuid,
          'UCHAR_MAX' => *::UCHAR_MAX,
          'CHILD_MAX' => *::CHILD_MAX,
          'fopen' => *::fopen,
          'Tie::' => *{'::Tie::'},
          'SIGQUIT' => *::SIGQUIT,
          'USHRT_MAX' => *::USHRT_MAX,
          'SIG_IGN' => *::SIG_IGN,
          'tzset' => *::tzset,
          'sigprocmask' => *::sigprocmask,
          'SEEK_SET' => *::SEEK_SET,
          'strtoul' => *::strtoul,
          'CLK_TCK' => *::CLK_TCK,
          'F_SETLKW' => *::F_SETLKW,
          'F_UNLCK' => *::F_UNLCK,
          'S_IXOTH' => *::S_IXOTH,
          'PARMRK' => *::PARMRK,
          'UNIVERSAL::' => *{'::UNIVERSAL::'},
          'pathconf' => *::pathconf,
          'FLT_ROUNDS' => *::FLT_ROUNDS,
          'L_tmpname' => *::L_tmpname,
          'longjmp' => *::longjmp,
          'EISCONN' => *::EISCONN,
          'DBL_MANT_DIG' => *::DBL_MANT_DIG,
          '_<perlmain.c' => *{'::_<perlmain.c'},
```

When you “use in” a perl module, everything that it exports get copied into main:: (usually via Exporter) which takes time and wastes memory

You can avoid this by doing

use Net::SSLeay ();

instead of

use Net::SSLeay;

PRACTICALLY FREE GAINS

- Switching all the *use X;* statements to *use X ();* had the following results:
  - The program started up 12% faster.
  - The program used 8% less memory while running.
- **The more *use* statements your program has, the better your returns will be.**
- You will have to explicitly call function ie Data::Dumper::Dumper instead of Dumper

SUMMARY

- Use NYTProf
- Come up with a new approach or clean it up using what your know or the tips here:
  - NCLARK's presentation titled "When perl is not quite fast enough." It has been one of the most useful bits perl documentation I have ever read: http://www.flirble.org/~nick/P/Fast_Enough/
- Verify your results with time and Benchmark